Consistency in UML and B multi-view specifications

Dieu Donné Okalas Ossami, Jean-Pierre Jacquot, Jeanine Souquières

Consistency in UML and B multi-view specifications

Dieu Donné Okalas Ossami, Jean-Pierre Jacquot, and Jeanine Souquières

LORIA - Université Nancy 2 - UHP Nancy 1
Campus scientifique, BP 239
54506 Vandœuvre-lès-Nancy Cedex - France
Email: {okalas,jacquot,souquier}@loria.fr

Abstract. We present the notion of *consistency relation* in UML and B multi-view specifications. It is defined as a semantic relation between both views. It provides us with a sound basis to define the notion of *development operator.* An operator models a development step; it separates the design decisions from their expression in the specification formalisms. Operator correctness is defined as a property which guarantees that the application of an operator on a consistent specification state yields a consistent new state. An operator can be proven once and for all to be correct. A classical case-study, the Generalized Railroad Crossing (GRC), demonstrates how the different notions can be put in practice to provide specifiers with a realistic development model.

Keywords: consistency, verification, operator, multi-view, UML, B.

1 Motivations

It has been recognized for a long time that the development of quality software depends crucially on the quality of the initial specification. Currently, there are two mains streams of specification languages: graphical notations such as UML which are very effective for the discussion between users and developers but are poor for formal verifications, and mathematical notations such as B which are effective for verification but very poor for discussion. Our aim is to design a framework where both kinds of notations can be used together to fulfill the needs of all the people involved.
The approach aims to capitalize on existing languages rather than to define a new one. This allows us to reuse the efforts that have been made in the production of industrial tools such as Rational Rose[1] or ArgoUML[2] for the edition of UML diagrams, and such as *Atelier B* [24], *B−Toolkit* [3], or *B4Free* [4] for the formal verification of specifications.
Our approach builds on the works made on the transformation between UML and B. [7, 10, 11, 13, 14, 16, 22] have defined precise sets of transformation rules to generate a B specification from UML diagrams. These works allow specifiers to check UML specifications by using B-based theorem provers. On the other way, [5, 6, 25, 26] define rules to generate UML diagrams from a B specification. These works allow specifiers to present users with a "readable" specification in order to ease the discussion and agreement on what the planned software is supposed to do.

[1] http://www-306.ibm.com/software/rational/
[2] http://www.argouml.tigris.org

Currently, case tools based on transformation rules, such as those proposed in [12, 23, 27], work with transformations in one direction. This is a consequence of the difficulty to integrate formalisms founded on different paradigms: object theory on the one hand and set theory on the other hand. This situation introduces a new problem: the specification development process is constrained in a highly unrealistic way. Let us suppose a specifier writes a first specification in UML; he then transforms it in B and checks it with the prover; likely, he will need to edit the B specification to discharge the proof. How can the changes be retrofitted in the UML design? The problem is the same in the other direction where the check will consist in a validation with users. Using a "reverse" set of transformation rules is not realistic since the result may lead to a specification very far from the original.
More generally, the transformation approach makes impossible opportunistic strategies where the specifier chooses at some time to focus his work on structural design with UML and at some other time to define formal properties with B, without any predefined order. Another approach consists in integrating formal definitions like data-types in UML state diagrams [2].
In our model, a specification is defined as a couple $\langle\ SpecUML,\ SpecB\ \rangle$ where *SpecUML* is a set of UML diagrams and *SpecB* a set of B machines. Both parts are views of the *same* specification. The development of a multi-view specification is modeled as a sequence of applications of *operators* [18]. An operator models a development technique by separating the design decisions from their impact on the UML and B parts. In practice, application of an operator makes both views evolve simultaneously through the application of specific editing actions on each part while ensuring that both parts are kept consistent. The notion of consistency is then central to our model. It gives a precise meaning to the notion of multi-view specification. It provides us with the formal tool to define the correctness of operators.

The paper is organized as follows. Section 2 explicits the concept of operator and the consistency relation. Section 3 introduces the case study of the generalized railroad crossing (GRC). Section 4 presents an example of operator : $Refine{-}Data$. Section 5 presents the application of operators on the case study. Section 6 gives proofs on the preservation of the consistency relation with respect to the applied operators. Section 7 concludes the paper.

2 Operators and multi-view consistency

2.1 Framework for operators

The development of a UML and B multi-view specification $Spec = \langle SpecUML,\ SpecB\rangle$ is done by the application of operators, making parallel couples of specifications $\langle SpecUML,\ SpecB\rangle$ evolve. An operator is composed of two parts: one working on *SpecUML* and the second on *SpecB*. These parts constitute language specific operators, denoted by $\mathcal{O}_{UML}$ and $\mathcal{O}_B$.
An operator has application conditions, ensuring the preservation of a global property of the whole specification *Spec*. To make the couple of specifications $\langle SpecUML, SpecB\rangle$ evolve, we have to determine the kind of changes we want to achieve as well as their

location. This corresponds to the selection of an operator. To guide the development, the *"Remain To Be Done"* clause provides information about which operators can be applied next in order to terminate a given development process. This means that operators must support the following combination features:
• *Recursion*. An operator can call itself.
• *Sequencing*. Operators can be sequenced to fire one after another.

2.2 Operator template

Operator : *operator_Name*

Description. Natural language description of the purpose and effects of the operator.

Parameters.
- **In.**
  ⟨*PARAM_Name* : *TYPE_PARAM*⟩*
- **Result.**
  ⟨*PARAM_Name* : *TYPE_PARAM*⟩*

Application conditions.

- Related to *SpecB*
  ⟨*COND_B*⟩*
- Related to *SpecUML*
  ⟨*COND_UML*⟩*

Definition.

- **Context.** ⟨*context*⟩
- ⟨*OPERATOR_DEF*⟩

Remains To Be Done. ⟨*To Do Next*⟩

Fig. 1. Operator template

The standard template for the definition of an operator is composed of various clauses. Each clause is optional except the first one. Each clause is described as follows:

1. **Parameters**. Determines parameters of the operator which are of two kinds, *In* and *Result*. They are both optional.
   - **In**. Designates elements needed to calculate the *Result* representations.
   - **Result**. Designates elements *created* by the operator. The *Result* parameters that are not included, default to the *In* parameters.

2. **Application conditions**. Defines the conditions which specify when the operator can be applied. Two kinds of conditions are identified : conditions related to UML (CONd_UML) and conditions related to B (COND_B).

3. **Definition**. Consists of:

– **Context**. Determines the element(s) the operator is applied to.
– $\langle$*OPERATOR_DEF*$\rangle$. Determines the sequence of operators to be applied. *OPERATOR_DEF* is given by the following grammar:

OPERATOR_DEF ::= $\langle \mathcal{O}_{UML},\ \mathcal{O}_B \rangle$
| OPERATOR_APP
| OPERATOR_DEF [; OPERATOR_DEF]*
| **IF** $\langle$ *COND* $\rangle$ **THEN** OPERATOR_DEF [; OPERATOR_DEF]*

where:
- *COND* denotes a condition on the context or the parameters.
- *OPERATOR_APP* denotes an operator's application consisting of the name of the operator and its parameters.

4. **Remains To Be Done.** Indicates which part of the specification has to be defined next.

2.3 Multi-view consistency relation

We consider the question of how the application of an operator has to be constrained so that its application on a current consistent specification state $\langle$*SpecUML*, *SpecB*$\rangle$, yields a consistent specification state $\langle$*SpecUML*$'$, *SpecB*$'\rangle$. Let us denote $\mathcal{R}_{el_C}$ the consistency relation between *SpecUML* and *SpecB*.

Let $\mathbb{T}_{U\rightarrow B}$ be the set of UML to B transformation rules [9, 17] which associate each UML artifact with one or more B artifacts. These transformations are relative to UML 1.x [19]. $\mathcal{R}_{el_C}$ is defined as a conjunction of four conditions:

1 *Syntactic conformance.* It states that both *SpecUML* and *SpecB* must be well-formed. It ensures that the specification conforms to abstract syntax specified by the meta-model, i.e. UML meta-model or B abstract syntax tree. Let $\mathcal{WF}$(*SpecUML*) and $\mathcal{WF}$(*SpecB*) be two predicates defining if a UML and a B specifications are well-formed.

2 *Local consistency.* It requires that both specifications must be internaly consistent. That means they do not contain contradictions, but they could be incompletely defined. We write it *consistent(SpecUML)* and *consistent(SpecB)*.

The global consistency is defined with respect to UML to B transformation rules designed by Meyer, Souquières and Ledang [9, 17].

3 *Elements traceability.* It states that for any elements of *ID*(*SpecUML*), e_U, that can be transformed by a rule *T*, there exists in *ID*(*SpecB*) a set of artifacts $\{e_B\}$ resulting from the application of *T* to e_U.

4 *Semantic preservation.* It states that any statement ϕ satisfying the semantics of *SpecUML* must satisfy *SpecB*. The semantics of *SpecUML* is defined as $\mathbb{T}_{U\rightarrow B}$ (*SpecUML*). This means that UML artifacts that have no B semantics defined in $\mathbb{T}_{U\rightarrow B}$ are not concerned by the consistency relation $\mathcal{R}el_C$. This has important implications throughout the verification process. For example, it is well known that checking pairwise integration of a set of software specifications is only possible if one is able to transform them into a semantic domain supported by tools. B is our semantic domain and any UML statement that has no B formalization cannot be verified in our framework.

We use the B theorem prover to prove that a statement ϕ holds in *SpecB* (condition (2)) and due to *condition (3)*, we derive the consistency of *SpecUML*, and therefore the consistency of the multi-view specification *Spec*.

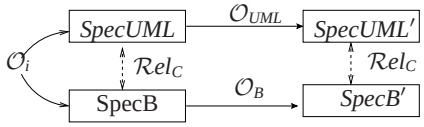


Fig. 2. UML and B consistency relation

Formally, the $\mathcal{R}el_C$ relation is defined as follows:

Definition 1 (Consistency relation)

SpecUML $\mathcal{R}el_C$ *SpecB* :

(1) $\mathcal{WF}(SpecUML) \wedge \mathcal{WF}(SpecB)$

(2) *consistent(SpecUML)* $\wedge$ *consistent(SpecB)*

(3) $\forall\ e_U.(e_U \in ID(SpecUML_{|_{\mathbb{T}_{U\rightarrow B}}})$ [a] $\Rightarrow$
$\exists\ \{e_B\},\ T.(\{e_B\} \subseteq ID(SpecB) \wedge T \in \mathbb{T}_{U\rightarrow B} \wedge T(e_U) = \{e_B\}))$

(4) $\forall\ \phi.(\mathbb{T}_{U\rightarrow B}(SpecUML)^b \vDash \phi \Rightarrow SpecB \vDash \phi))$

[a] $SpecUML_{|_{\mathbb{T}_{U\rightarrow B}}}$ denotes the restriction of *SpecUML* to elements for which there is a transformation rule to B defined in $\mathbb{T}_{U\rightarrow B}$

[b] $\mathbb{T}_{U\rightarrow B}$(SpecUML) denotes the application of the set of UML to B transformation rules on *SpecUML*

3 A case study

The evolution of the specification and the verification of the consistency relation described in section 2.3 will be applied to the generalized railroad crossing example, called *GRC* in the sequel. We give a short description of the problem and an abstract specification on which the refinement can be introduced. Fig. 3 illustrates the structure of the *GRC* extracted from [21]. The system to be modeled consists of a gate, a controller and trains at a railroad crossing.

The railroad crossing lies in a region of interest *R*. Trains travel in one direction through *R*, and only one train per track is allowed to be in *R* at any moment. Sensors indicate when a train enters or exits the region *R*. For space and clarity reasons, we do not present in details the *GRC* problem, but only details which are relevant to illustrate our approach.

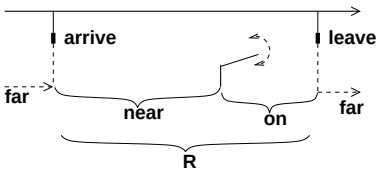


Fig. 3. The generalized railroad crossing

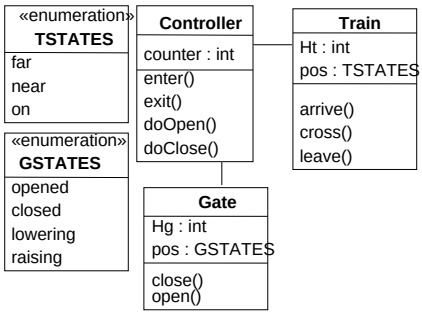
We will describe the development of the system step by step, starting with the UML specification which identifies some important entities. Note that we only focus on static aspects.

3.1 A first UML specification

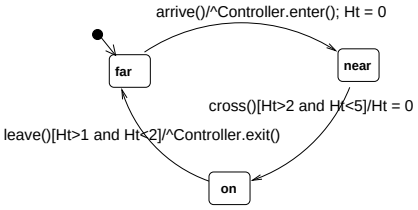
A *Train* may be in three states: *far*, *near* and *on*. The state of the train is determined by the information provided by sensors positioned on the track and by a clock. When a train leaves a region and enters another one, a signal is sent to the controller which reacts by sending appropriate signals to the gate. A train takes *2* to *5* time units to reach state *on* after it entered state *near*. It then leaves state *on* and therefore region *R* and reaches state *far* between *1* and *2* time units. Time information is stored in the variable *Ht*, which is initialized to *0* when a train enters state *near* and state *on*. The system must be safe: the gate must be down when trains reache state *on*. In order to have the gate closed when the fastest train reaches state *on*, the gate must be closed between *1* and *2* time units after trains entered region *R*.
The class *Train* is characterized by the following variables:

- *Ht*, which models the time taken by the train to reach each state,
- *pos*, which models the train states.

The class Train provides three methods, *arrive()*, *cross()* and *leave()*, for entering, crossing and leaving region *R*, respectively. The class diagram of the *GRC* and the behavior of the *Train* are presented in Fig. 4(a) and 4(b).



(a) Class diagram of the GRC



(b) State diagram of Train

Fig. 4. A first UML specification

3.2 The corresponding B specification

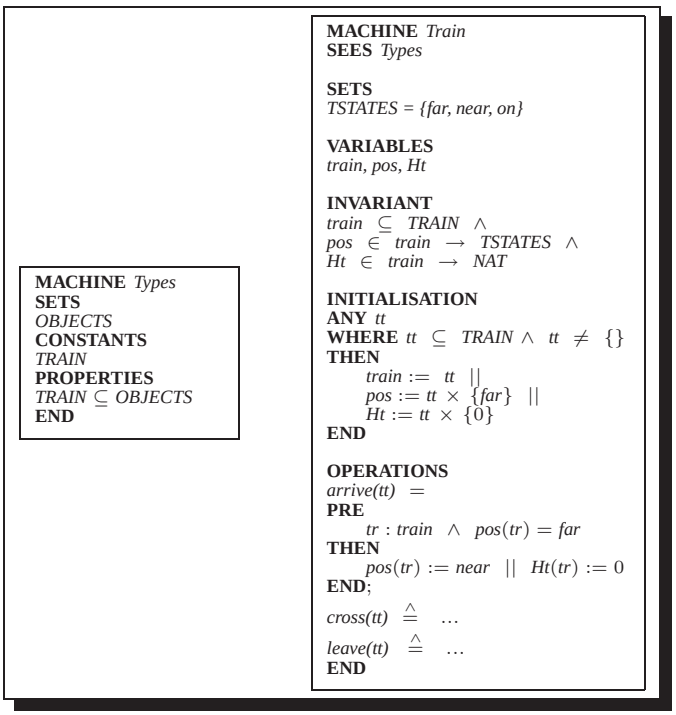


Fig. 5. Associated B Machines

Figure 5 represents the abstract specification of the class *Train* obtained by an automatic translation of the UML specification (cf. Fig. 4). Each class with name *Class* is represented by an abstract machine with the same name, as discussed in [16]. For each class *Class*, a set *CLASS* is introduced to represent all possible instances of *Class*. A variable $class \subseteq CLASS$ is used to identify current instances of *Class*. Attributes are modeled as functions from *class* to the attribute type as defined in the class. The type of functions reflects the participation and cardinality of the entity. Class operations are derived as B operations (e.g. *arrive*, *cross* and *leave* in the *Train* machine of Fig. 5) mirroring the syntactical structure of the associated state diagram. Operation parameters are typed and further constrained in the operation precondition. Operation bodies are automatically derived from transitions in the state diagram. In addition to machines representing classes, we introduce a special machine *Types*, which declares a number of shared sets or types. The others classes are derived in a similar way.

3.3 Improving the specification

Let's take the UML and B specification couple of Fig. 4(a) and 5 and consider that the user focuses his work on the B specification. He decides to observe more in details the behavior of the train in the state *near*, as described informally in Fig. 7 and graphically

illustrated in Fig. 6.

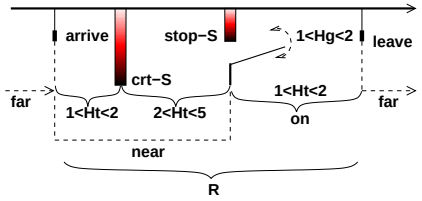


Fig. 6. Detailed GRC

If the train moves at great speed towards the crossing and arrives at point *crt-S* (critical state) in less than *2* time units, it must stop at *stop-S* (stop state). It then starts to move again, when the time variable *Ht* is greater than two time units. This is the time needed by the gate to be completely closed.

Fig. 7. Example of a critical property

The description of this property requires new variables, states, types, and constraints to be added to the initial specification of the train. This means that we have to improve the current couple of specifications so that it captures this new requirement. The refinement is an appropriate technique to express this critical property. For this purpose, we provide users with the *Refine-Data* operator, allowing to enrich the current specification in a stepwise manner. It also provides a way to strengthen invariants and to add details omitted in previous abstractions. The *Refine-Data* operator defined in this paper is used to replace some types and data in a specification by more concrete ones in order to increase efficiency or implementability. The replacement ends up in new entities, sets and constraints on the data space being introduced in the specification. Note that we do not attempt to provide a new definition of data refinement, rather we use the standard definition of refinement of state variables. From a practical perspective, we present the data refinement process as follows.

- First, concepts (e.g., refinement component, variables, types, classes, attributes, etc.) that form the basis for expressing properties are modeled. The *Refine-Data* operator is defined to act this role.
- Second, we consider concepts such as gluing invariants or additional constraints over data introduced previously to express logical links between concrete data and their abstract versions. This is achieved by using the *Model-Constraint* operator.

For naming UML and B model elements, we will consider the following notations:

- *ID* the set of all identifiers of the specification (*ID = ID(SpecB)* $\cup$ *ID(SpecUML)*).
- *CMP(SpecB)* $\subseteq$ *ID(SpecB)* the finite set of B components (machine, refinement, implementation) names appearing in *SpecB*.
- *CLASS(SpecUML)* $\subseteq$ *ID(SpecUML)* the finite set of class names appearing in *SpecUML*.
- *ATT*(C) the finite set of attributes of a class $C \in$ *CLASS(SpecUML)*.
- *DATA*(Ma) the finite set of data, such as variables and constants, appearing in a B component $Ma \in$ *CMP(SpecB)*.

4 An example of operator: Refine-Data

Operator: Refine-Data
Description. This operator provides a scheme to refine data, replacing some types and data in a specification by more concrete ones in order to support the addition of functional details, to increase efficiency or implementability. Users must designate:

- a B component *Ma* to which the data to be refined belongs,
- a variable *v: S* the user wants to refine,
- a state s_i the user wants to precise, if the type *S* of *v* is a set of states $\{s_0, ..., s_i, ..., s_n\}$
- a set of concrete versions $\{s_{r_i}, ..., s_{r_j}\}$ the user wants to replace s_i with. If *S* is an abstract set, the user will give explicit values $\{s_{r_i},...s_{r_j}\}$ to it.

The following modifications are made to $\langle SpecUML,\ SpecB\rangle$:

In *SpecB*

1. If there is no already existing refinement Ma_r of Ma[3] (denoted by $Ma \sqsubseteq Ma_r$), a refinement Ma_r is automatically introduced. It models the following elements:

   (a) a *REFINES Ma* clause immediately after its header, identifying the single component *Ma* that it refines,

   (b) a set:
   i. $S_r = S \cup \{s_{r_i}, ..., s_{r_j}\}$ that refines the more abstract set *S* ($S \sqsubseteq S_r$) if *S* is an enumerated set. Note that S_r is composed of new state values, as well as of all values[4] in *S* or
   ii. $S_r = \{s_{r_i},...s_{r_j}\}$ if the to refined set *S* is abstract,

   (c) a set of state variables $\{v_{r_0}, ..., v_{r_i}, ..., v_{r_n}\}$ which take their values in S_r, if *v* must be refined by several variables,

   (d) a comment line $<$ *To do J(*v, v_{r_i}*)* $>$ denoting the location to be replaced with the *gluing invariant* that relates the abstract state variable *v* and the concrete state variables $\{v_{r_0}, ..., v_{r_i}, ..., v_{r_n}\}$ and extra constraints (refinement conditions).

2. If there is already a refinement Ma_r of *Ma*, two cases can occur:

   (a) in the first case, additional local types, data and extra constraints may be added to Ma_r in a similar way than 1b and 1c,
   (b) in the second case, Ma_r may be precised by means of a serie of refinements in a similar way than in the case 1. The refinement process will also iterate.

[3] This is the case when the operator is applied for the first time on *Ma*.
[4] They are renamed in S_r in order to satisfy B naming conventions. For instance, s_i in *S* is renamed by s_iR in S_r.

3. If the type of v is a predefined type, no new types are introduced.

In *SpecUML*

1. **In the class diagram**

   (a) a refinement component Ma_r corresponds to the class Ma_r,

   (b) a B variable v_r in Ma_r corresponds to an attribute v_r in the class that represents the refinement component Ma_r. The type and initial value of this attribute correspond to the type invariant and initialization substitution of the corresponding B variable,

   (c) a set S_r in B corresponds to an enumerated type in UML or in OCL,

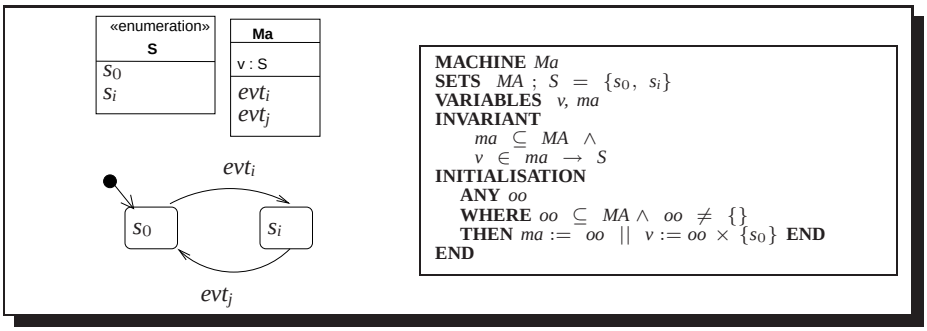
   (d) the *REFINES* clause is modeled by an abstraction/refinement association with a $<<refines>>$ stereotype,

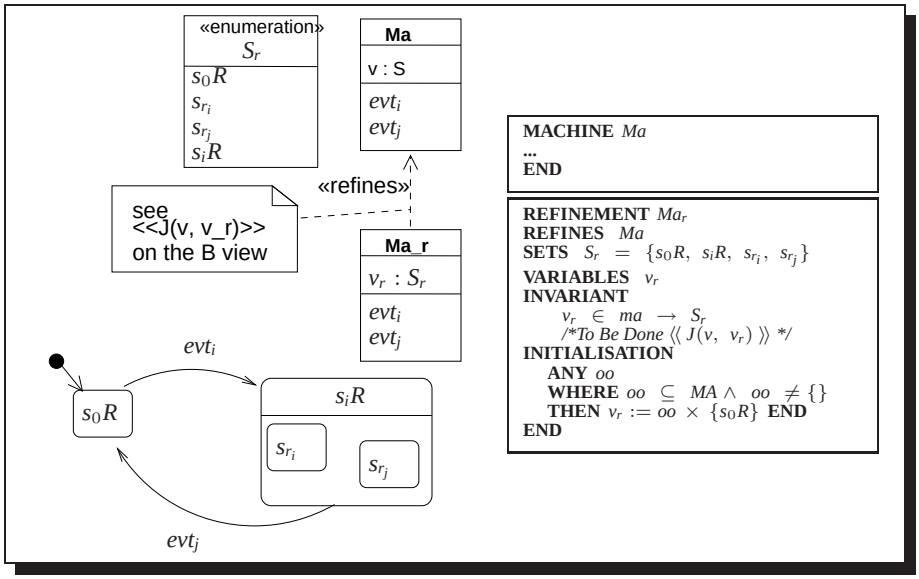
   (e) the comment line $<$ *To do J*(v, v_{r_i}) $>$ leads to the creation of a comment note in the class diagram, referencing the refinement link between the abstract and the refinement class.

2. **In the state diagram**

   (a) For a state s_{r_i} that refines a state s_i, a super-state s_i with sub-state s_{r_i} is drawn by nesting in the state diagram attached with the class *Ma*.

   (b) A set of state values $S_r = \{s_0R, ..., s_iR, s_{r_i}, ..., s_{r_j}, ..., s_nR\}$ refining an abstract set of state values S ($S \sqsubseteq S_r$) leads to the generation of a state diagram as shown below.





Result of the Refine-Data operator application

Parameters.
In

- *Ma* : identifier
- *v* : identifier
- $[s_i : State]$[5]
- $[\{s_{r_i}, ..., s_{r_j}\} : States]$

Result

- Ma_r : identifier
- v_r : identifier
- S_r : identifier

Application conditions.

1. Related to *SpecB*
   - $Ma \in CMP(SpecB) \wedge Ma$ ::= MACHINE | REFINEMENT
   - $s_i \in$ S
   - v : S $\wedge$ $v \in DATA(Ma)$
   - $\forall\ s_k.(\ s_k \in \{s_{r_i}, ..., s_{r_j}\} \Rightarrow s_k \notin s_i)$

2. Related to *SpecUML*
   - $\exists\ C.(C \in CLASS(SpecUML) \wedge$ C $\mapsto$ Ma)
   - $\exists\ a.(a \in ATT(C) \wedge (a \mapsto v))$

[5] $[x]$ denotes that x is optional.

– $\exists\ T.(T \in TYPE(SpecUML) \wedge \mathrm{T} \mapsto \mathrm{S})$

Definition.

Context: *Ma*
IF (Ma ::= MACHINE $\vee$ Ma ::= REFINEMENT) $\wedge$
(v $\in$ DATA(Ma) $\wedge$ (S ::= AbstractSet $\vee$ S ::= EnumeratedSet))
THEN

$\mathcal{O}_{UML}$	$\mathcal{O}_B$
$(AddClass(Ma_r)\ ;\ AddDependency(Ma,\ Ma_r,\ «refines»))\ ;$ $AddType(S_r,\ \{s_{r_i}, ..., s_{r_j}\})^*\ ;$ $AddAttribute(Ma_r,\ v_r)^*$	$AddRefinement(Ma,\ Ma_r)$ $AddSet(S_r,\ \{s_{r_i}, ..., s_{r_j}\})^*\ ;$ $AddVariable(Ma_r,\ v_r)^*$

;

IF (Ma ::= MACHINE $\vee$ Ma ::= REFINEMENT) $\wedge$
(v $\in$ DATA(Ma) $\wedge$ S ::= PredefinedType)
THEN

$\mathcal{O}_{UML}$	$\mathcal{O}_B$
$(AddClass(Ma_r)\ ;\ AddDependency(Ma,\ Ma_r,\ «refines»))\ ;$ $AddAttribute(Ma_r,\ v_r)^*$	$AddRefinement(Ma,\ Ma_r)$ $AddVariable(Ma_r,\ v_r)^*$

;

IF Ma ::= REFINEMENT $\wedge$
v $\in$ DATA(Ma) $\wedge$ (S ::= AbstractSet $\vee$ S ::= EnumeratedSet) $\wedge$
$\exists\ Ma_x.(Ma_x \in ID(Spec) \wedge Ma\ \sqsubseteq\ Ma_x)$
THEN

$\mathcal{O}_{UML}$	$\mathcal{O}_B$
$AddType(S_r,\ \{s_{r_i}, ..., s_{r_j}\})^*\ ;$ $AddAttribute(Ma_x,\ v_r)^*$	$AddSet(S_r,\ \{s_{r_i}, ..., s_{r_j}\})^*\ ;$ $AddVariable(Ma_x,\ v_r)^*$

;

IF Ma ::= REFINEMENT $\wedge$
v $\in$ DATA(Ma) $\wedge$ S ::= PredefinedType $\wedge$
$\exists\ Ma_x.(Ma_x \in ID(Spec) \wedge Ma\ \sqsubseteq\ Ma_x)$
THEN

$\mathcal{O}_{UML}$	$\mathcal{O}_B$
$AddAttribute(Ma_x,\ v_r)^*$	$AddVariable(Ma_x,\ v_r)^*$

Remains To Be Done. The introduced variable can be improved:
• Invariant and initialization comment lines have to be replaced by concrete constraints using for example the *Model-Constraint* operator.

5 Application to the case study

Let's take the couple of specifications of Fig. 4 and Fig. 5 and enrich it with new variables *posR* and *HtR* in order to model the property of Fig. 7. We decide to refine the *Train* machine, using the *Refine-Data* operator. The train machine and its related UML class and state diagram are interdependent representations. As one changes, the other

one undergoes changes too. So, we instantiate twice the *Refine-Data* operator in order to introduce variables *posR* and *HtR*. The instantiation of this operator requires to set the actual parameters as shown in Fig. 8(a) and 8(b).

Parameters
In
(*Train, pos, near,* {crt_S, stop_S}
Results
Train_R, posR, TSTATES_R

(a) First instantiation to introduce the variable *posR*

Parameters
In
(*Train, Ht*)
Results
HtR

(b) Second instantiation to introduce the variable *HtR*

Fig. 8. Two instantiations of the *Refine−Data* operator

Fig.9 illustrates the result of the instantiation of the *Refine−Data* operator on the *Train* machine, where variables *posR* and *HtR* are introduced one after the other.

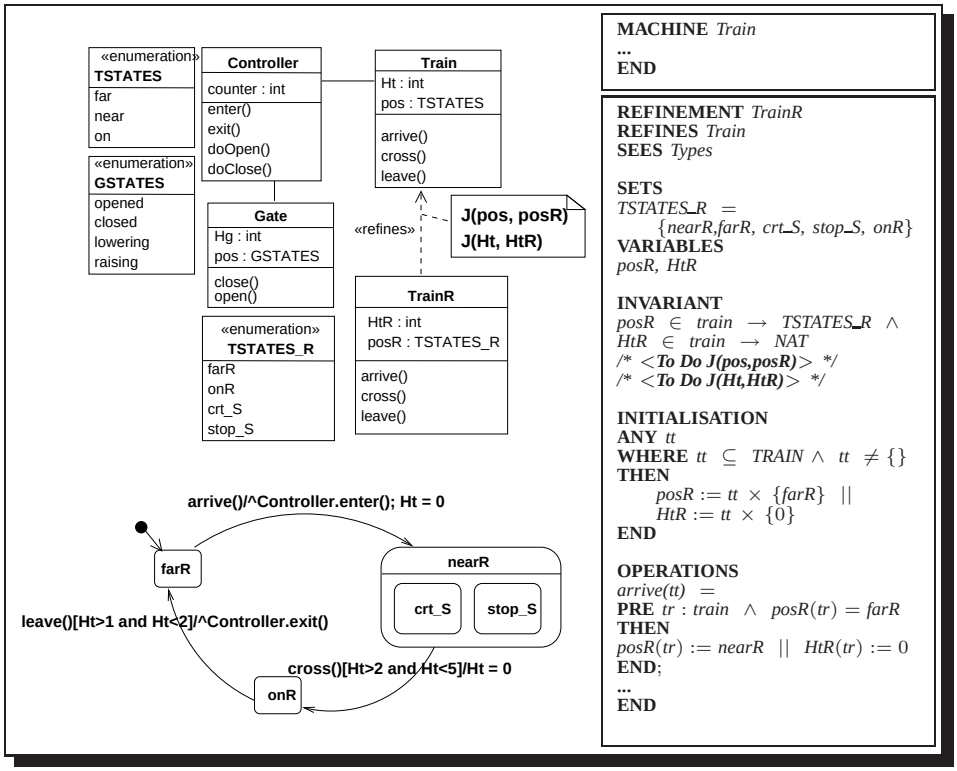


Fig. 9. Application of the *Refine−Data* operator on the train machine

One of the most important steps when refining specifications in B is formulating *gluing invariants* that relate concrete variables with their abstract versions. We assume that the data refinement process ends with formulating invariants over variables and types

previously introduced by the *Refine-Data* operator in the first step. This is achieved by the *Model-Constraint* operator as indicated in the **Remains To Be Done** clause of the *Refine−Data* operator. Once the *Model-Constraint* operator has been applied, we can move on to deal with the consistency checking. For space and clarity reasons, we do not present the definition of the *Model-Constraint* operator in this paper.

For the *Model-Constraint* operator to work, users have to write the invariant I to be added, and to designate the component to which this constraint has to be introduced. The application ends with new constraints in B and OCL constraints or comment notes in UML. We give below one possible formulation of the gluing invariant (over variables *posR* and *HtR*, and their abstract versions *pos* and *Ht*) and constraints on the new functionality that can be given as parameter when instantiating the *Model-Constraint* operator.

Parameters
In
TrainR

$$
\begin{array}{l}
\textit{/* gluing invariants */} \\
\forall\, tr.(tr \;:\; train \;\Rightarrow \\
(\; posR(tr) \;=\; farR \;\Rightarrow\; pos(tr) \;=\; far \;) \wedge \\
(\; (\; posR(tr) \;=\; crt\_S \textit{ or } posR(tr) \;=\; stop\_S \;) \;\Rightarrow\; pos(tr) \;=\; near \;) \;\wedge \\
(posR(tr) \;=\; onR \;\Rightarrow\; pos(tr) \;=\; on \;) \;\wedge \\
(\; HtR(tr) \;:\; 0..5 \;\Rightarrow\; HtR(tr) \;=\; Ht(tr) \;) \wedge \\
\textit{/* constraints on the new functionality */} \\
(\; posR(tr) \;:\; \{\; crt\_S,\; stop\_S,\; onR\} \;\Rightarrow\; HtR(tr) \;<\; 5 \;) \;\wedge \\
(\; posR(tr) \;=\; crt\_S \textit{ or } posR(tr) \;=\; onR \;\Rightarrow\; HtR(tr) \;<\; 2 \;)\;)
\end{array}
$$

REFINEMENT *TrainR*
REFINES *Train*
SEES *Types*
SETS $TSTATES\_R = \{nearR,\; farR,\; crt\_S,\; stop\_S,\; onR\}$
VARIABLES *posR, HtR*
INVARIANT

$$
\begin{array}{l}
posR \;\in\; train \;\rightarrow\; TSTATES\_R \;\wedge \\
HtR \;\in\; train \;\rightarrow\; NAT \;\wedge \\
\textit{/* gluing invariants */} \\
\forall\, tr.(tr \;:\; train \;\Rightarrow \\
(\; posR(tr) \;=\; farR \;\Rightarrow\; pos(tr) \;=\; far \;) \wedge \\
(\; (\; posR(tr) \;=\; crt\_S \textit{ or } posR(tr) \;=\; stop\_S \;) \;\Rightarrow\; pos(tr) \;=\; near \;) \;\wedge \\
(posR(tr) \;=\; onR \;\Rightarrow\; pos(tr) \;=\; on \;) \;\wedge \\
(\; HtR(tr) \;:\; 0..5 \;\Rightarrow\; HtR(tr) \;=\; Ht(tr) \;) \wedge \\
\textit{/* constraints on the new functionality */} \\
(\; posR(tr) \;:\; \{\; crt\_S,\; stop\_S,\; onR\} \;\Rightarrow\; HtR(tr) \;<\; 5 \;) \;\wedge \\
(\; posR(tr) \;=\; crt\_S \textit{ or } posR(tr) \;=\; onR \;\Rightarrow\; HtR(tr) \;<\; 2 \;) \\
)
\end{array}
$$

INITIALISATION
...
END

Fig. 10. B refinement of the class Train

Fig. 10 shows the B specification of *TrainR* after the application of the *Constraint−Modeling* operator. Because existing OCL to B rules [11] are only defined for simple expressions, there is no creation of an OCL constraint for the introduced B invariant I.

6 Verification of the operator's correctness

In this section, we look at the correction aspect of the case-study. We show concretely how the definitions apply to the UML and B parts manipulated in the case-study. We also give some hints on how the correctness of the operators *Refine-Data* and *Model-Constraint* could be assessed.

6.1 Syntactic well-formedness

Both specifications must be checked for syntax and type correctness with their corresponding support tool. The B support tool we use for this case study, *atelierB*, confirms the well-formedness of the text shown in Fig. 10. The UML diagrams are also well-formed according to ArgoUML.

6.2 Internal consistency

The definition of operator correctness uses the strong hypothesis that each view in the initial state is internally consistent. While this condition is not much more than the well-formedness for the UML, it means full logical consistency for the B part.
The checking of *SpecB* follows the usual approach of the B method: to check initialization, to check pre and postconditions of operations with respect to the preservation of machine invariants, and to check inter-machine relations such as refinements. On the case-study, it is clear that the verification is done on two levels. The first level is the verification that the elements automatically introduced by the operator in *SpecB* are correct. The second level checks that the elements introduced by the user are consistent. In our case, the first level is mainly exemplified by the operator *Refine-Data*, while *Model-Constraint* is mostly about the second level.
SpecB has been submitted to the *atelierB*. All proof obligations generated by the `REFINEMENT` status of the `TrainR` have been discharged through the gluing invariant which was introduced by the application of the *Model-Constraint* operator. Figure 11 shows the summary of the verification printed by the tool.

```
 Project status
+-----------+----+-----+-----+-----+-----+-----+
| COMPONENT | TC | POG | Obv | nPO | nUn | %Pr |
+-----------+----+-----+-----+-----+-----+-----+
| Train     | OK | OK  | 0   | 4   | 0   | 100 |
| TrainR    | OK | OK  | 3   | 10  | 0   | 100 |
| Types     | OK |  -  |     |     |     |     |
+-----------+----+-----+-----+-----+-----+-----+
| TOTAL     | OK | -   | 3   | 14  | 0   | 100 |
+-----------+----+-----+-----+-----+-----+-----+
```

Fig. 11. Result of the verification of the B specification

6.3 Consistency between views

It is decomposed into the *elements traceability* and *semantics preservation* conditions. Let's consider:

- $\langle SpecUML,\ SpecB\rangle$ the specification couple of Fig. 4 and 5, respectively.
- $\langle SpecUML',\ SpecB'\rangle$ the specification couple of Fig. 9 and 10, resulting from the application of the operators on $\langle SpecUML,\ SpecB\rangle$. Note that Fig. 10 includes the machine *Train* of Fig. 9.
- $\mathbb{T}_{U\rightarrow B}$ the set of UML to B transformation rules by Meyer [15] and Ledang [8].

To check, we apply the transformation rules $\mathbb{T}_{U\rightarrow B}$ to *SpecUML′*. The interesting part of the B specification, the machines *Train** and *TrainR**, is given in Figure 12. It then proceeds by the verification of conditions 3 and 4 of $\mathcal{R}_{el_C}$.

MACHINE *Train**
SEES *Types*
SETS
TSTATES = {far, near, on}
VARIABLES
train, pos, Ht
INVARIANT
$train \subseteq TRAIN \ \wedge$
$pos \in train \rightarrow TSTATES \ \wedge$
$Ht \in train \rightarrow NAT$
INITIALISATION
ANY
tt
WHERE
$tt \subseteq TRAIN \wedge tt \neq \{\}$
THEN
$train := tt \ ||$
$pos := tt \times \{far\} \ ||$
$Ht := tt \times \{0\}$
END
OPERATIONS
arrive(tt) =
PRE
$tr : train \ \wedge \ pos(tr) = far$
THEN
$pos(tr) := near \ || \ Ht(tr) := 0$
END;
cross(tt) $\hat{=}$ …
leave(tt) $\hat{=}$ …
END

MACHINE *TrainR**
SEES *Types*
SETS
TSTATES_R = {farR,nearR, crt_S, stop_S, onR}
VARIABLES
trainR, posR,HtR
INVARIANT
$trainR \subseteq TRAINR \ \wedge$
$posR \in trainR \rightarrow TSTATES\_R \ \wedge$
$HtR \in trainR \rightarrow NAT$
INITIALISATION
ANY
tt
WHERE
$tt \subseteq TRAINR \wedge tt \neq \{\}$
THEN
$trainR := tt \ ||$
$posR := tt \times \{far\} \ ||$
$HtR := tt \times \{0\}$
END
OPERATIONS
arrive(tt) =
PRE
$tr : trainR \ \wedge \ posR(tr) = farR$
THEN
$posR(tr) := nearR \ || \ HtR(tr) := 0$
END;
cross(tt) $\hat{=}$ …
leave(tt) $\hat{=}$ …
END

Fig. 12. B specification obtained by applying transformation rules

Condition 3 is proved by verifying that $ID(\mathbb{T}_{U\rightarrow B}(SpecUML')) = ID(SpecB')$. This is asserted in two steps:

- all new names introduced by the operators are present. This is easily seen,
- condition 3 holds for $\langle SpecUML,\ SpecB\rangle$. This is true by construction, cf. subsection 3.2.

The verification of condition 4 is more complex. When we look at Figure 12, we can see the following differences between *SpecB* and $\mathbb{T}_{U\rightarrow B}(SpecUML')$:

1. *TrainR** is a machine and is not related to *Train** by a refinement relation.
2. The machine *TrainR** introduces a new variable *trainR*[6] which is a subset of the set *TRAINR* representing possible instances of the class *TrainR*. *trainR* and *TRAINR* do not appear in *SpecB′*. As a consequence, variables *posR* and *HtR* in the machine *TrainR** which are modeled as functions from current instances set (*train*) to the corresponding type (STATES_R and *NAT* respectively), have now different domains.
3. the UML abstraction/refinement dependency is not modeled,
4. the added invariants in the machine *TrainR* of *SpecB′* do not appear since they have been represented as a comment note in *SpecUML′*.

So, to establish the property, we have to prove that the machine *TrainR** is a refinement of the machine *Train**. Concretely, we must find an abstraction function, ρ, defined as follows. Let us consider:

- S_{Ma_r} and S_{Ma} the sets of states of Ma_r and Ma respectively,
- Evt_{Ma_r} and Evt_{Ma} the sets of events of state machines of Ma_r and Ma respectively,
- $Trans_{Ma_r}$ and $Trans_{Ma}$ the sets of transitions of state machines of Ma_r and Ma respectively,

$\rho : Ma_r \rightarrow Ma$ is an abstraction relation which is a function from $S_{Ma_r} \cup Evt_{Ma_r} \cup Trans_{Ma_r}$ to $S_{Ma} \cup Evt_{Ma} \cup Trans_{Ma}$ and which maps

- Each state s_r of Ma_r to a state $\rho(s_r)$ of Ma,
- Each event e_r of Ma_r to an event $\rho(e_r)$ of Ma and
- Each transition t_r of Ma_r to a transition $\rho(t_r)$ of Ma.

Such that

- $\rho(s_{init_{Ma_r}}) = s_{init_{Ma}}$
- $Evt_{Ma}(\rho(t_r)) = \rho(Evt_{Ma_r}(t_r))$
- $source_{Ma}(\rho(t_r)) = \rho(source_{Ma_r}(t_r)) \wedge target_{Ma}(\rho(t_r)) = \rho(target_{Ma_r}(t_r))$

ρ is an abstraction function equivalent to a B refinement if the following properties hold:

1. $\forall\ s.\exists\ s_r.(s \in S_{Ma} \wedge s_r \in S_{Ma_r} \wedge \rho(s_r) = s) \wedge$
2. $\forall\ s.\exists\ t.(\ s \in S_{Ma} \wedge t \in Trans_{Ma} \wedge Evt_{Ma}(t) = e \wedge source_{Ma}(t) = s \Rightarrow$
   $(\forall\ s_r.(s_r \in S_{Ma_r} \wedge \rho(s_r) = s \Rightarrow \exists\ t_r.(t_r \in Trans_{Ma_r} \wedge$
   $\rho(Evt_{Ma_r}(t_r)) = e \wedge$
   $\rho(t_r) = t \wedge$
   $source_{Ma_r}(t_r) = s_r$
   $)))$
   $)$

The first condition states that every state *s* of an abstraction *Ma* has some corresponding states of its refinement Ma_r. The second states that every event *e* which has an abstract transition from some state *s* has also a corresponding concrete transition from each corresponding state.

[6] for modeling effective instances of the class *TrainR*

These conditions ensure that all properties expressed in Ma_r hold in the abstraction Ma and therefore the semantic preservation criteria is ensured. Actually, this condition is similar to the preservation of precondition requirement of B refinement.
The definition of ρ on our case study is as follows:

ρ = {*farR* $\mapsto$ *far*, *stop_S* $\mapsto$ *near*, *crt_S* $\mapsto$ *near*, *onR* $\mapsto$ *on*} $\cup$
{*arrive* $\mapsto$ *arrive*, *cross* $\mapsto$ *cross*, *leave* $\mapsto$ *leave*} $\cup$
{*(farR, arrive, nearR)* $\mapsto$ *(far, arrive, near)*, *(nearR, cross, onR)* $\mapsto$ *(near, cross, on)*,
(onR, leave, farR) $\mapsto$ *(on, leave, far)*}

It is easily verified that the preceding properties holds by considering the gluing invariant.

7 Conclusion and Future work

Combining UML notations and the B method is important for the use and the acceptance of formal methods as part of the development of high quality systems. We propose a framework allowing to define development operators making evolve UML and B multi-view specifications. The approach is not based on the application of transformation rules from UML to B or B to UML, but on the development of both specifications in an incremental way by applying operators. Operators enable the specifier to focus on methodological issues before addressing technical details related to each specification language.
We have proposed a definition of the consistency relation between both views of a specification expressed with UML and B, and two consecutive development states. The verification of the consistency is done once for all for each operator when defining them, relatively to a set of UML to B systematic transformation rules. It is partly automated and supported by the B prover.
As the case study shows, our approach does not pretend to automate the entire development of the specification. Technical and tedious syntactical details are taken care of by the operators but the design of important properties is still the specifier's responsibility.

An implementation of this framework with some operators is under development. It is an extension of the *ArgoUML+B* [12] platform, allowing to automatically transform some UML diagrams to B specifications (*ArgoUML+B* is based on the *ArgoUML*[7] project, dedicated to the edition and design of UML diagrams). This extension includes *SmartTools* [1, 20] to dynamically represent B specifications as instances of the B AST (abstract syntax tree), taking into account the multi-view specification.
We are looking at developing a library of useful operators. We have already identified and defined some restructuring operators such as modeling abstraction of generic classes from existing classes. We also need operators for the specification of system behaviours.

[7] http://www.argouml.tigris.org

References

[1] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, and C. S. Coen. SmartTools: a development environment generator based on XML technologies. In *In XML Technologies and Software Engineering, Toronto, Canada. ICSE'01, ICSE workshop proceedings*, 2001.

[2] C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. In *FASE'2003 - Fundamental Approaches to Software Engineering*, volume 2621 of *LNCS*, pages 341–355. Springer-Verlag, 2003.

[3] Oxford(UK) B-Core(UK) Ltd. *B-Toolkit User's Manual*. 1996.

[4] ClearSy. http://www.b4free.com/index.php.

[5] F. Houda and Stephan Merz. Transformation de spécifications B en diagrammes UML. In *Proceedings of AFADL'04, Besançon (Fr)*, 2004.

[6] A. Idani and Y. Ledru. Object Oriented Concepts Identification from Formal B Specifications. In *9th Int.Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), Linz (AT)*, 2004.

[7] R. Laleau and F. Polack. A Rigorous Metamodel for UML Static Conceptual Modelling of Information Systems. In *Advanced Information Systems Engineering. 13th Int. Conf., CAiSE 2001, Proceedings*, volume 2068 of *LNCS*, pages 402–416. Springer, 2001.

[8] H. Ledang. *Traduction Systématique de Spécifications UML vers B.* PhD thesis, LORIA -Université Nancy2, novembre, 2002.

[9] H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. - *ASE2001: 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, November*, 2001.

[10] H. Ledang and J. Souquières. Integrating Formalizing UML Behavioral Diagrams with B. *Workshop on Integration and Transformation of UML models, Malàga (S)*, 2002.

[11] H. Ledang and J. Souquières. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. In *Proceedings of APSEC 2002 IEEE Computer Society*, 2002.

[12] H. Ledang, J. Souquières, and S. Charles. ArgoUML+B : Un outil de transformation systématique de spécifications UML vers B. In *Proceedings of AFADL'03, Rennes (Fr)*, 2003.

[13] R. Marcano and N. Levy. Transformation rules of OCL constraints into B formal expressions. In Jürjens, Cengarle, Fernandez, Rumpe, and Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 155–162, 2002.

[14] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors, *Workshop on Consistency Problems in UML-based Software Development. Workshop Materials*, pages 91–105, 2002.

[15] E. Meyer. *Développements formels par objets: utilisation conjointe de B et d'UML.* PhD thesis, LORIA -Université Nancy2, mars, 2001.

[16] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. *FM'99: World Congress on Formal Methods in the Development of Computing Systems, Toulouse (Fr)*, 1999.

[17] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. *FM'99: World Congress on Formal Methods in the Development of Computing Systems, Toulouse (Fr)*, 1999.

[18] D. Okalas Ossami, J. Souquières, and J.-P. Jacquot. Opérations de construction de spécifications multi-vues UML et B. In *Proceedings of AFADL'04, Besançon, France, June 16-18*. INRIA, 2004.

[19] OMG. Unified modeling language specification, version 1.5, March 2003. available from hhtp::://www.omg.org.

[20] D. Parigot and C. Courbis. avaible at : http://www-sop.inria.fr/smartool/.

[21] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussine, and A. Petit. *Vérification de logiciels -Techniques et outils du model-checking-*. Paris,Vuibert, 1999. ISBN 2- 7117-8646-3.

[22] C. Snook, M. Butler, and I. Oliver. Towards a UML profile for UML-B. Technical report, DSSE-TR-2003-3, Electronics and Computer Science, University of Southampton, 2003.

[23] C. Snook and M. Buttler. U2B: a tool for combining UML and B. Avaible at http://www.ecs.soton.ac.uk/ cfs/U2Bdownloads/.
[24] STERIA. *Manuel de référence du langage B.* -ClearSy-, novembre, 1998.
[25] B. Tatibouet, A. Hammad, and J.-C. Voisinet. From an abstract B specification to UML class diagrams. In *2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002)*, pages 5–10, 2002.
[26] B. Tatibouet and J.-C. Voisinet. Generating statecharts from B specifications. In *16th International Conference Software & Systems Engineering and their applications (ICSSEA'2003)*, Paris (Fr), 2003.
[27] B. Tatibouet and J.C. Voisinet. jBtools and B2UML : a plateform and a tool to provide a UML class diagram since a B specification. In *ICSSEA : 14th International Conference on Software and Systems Engineering and Their Applications, Paris (Fr)*, volume 2, 2001.